Programming Project 1

Solving the 8-puzzle problem using A* search algorithm

Report

<u>Members:</u>

Vasu Tiwari (801254277)

Akshdeep Singh Rajawat (801208232)

Rishabh Mehta (801257231)

Contents

1. Problem Statement

Implementing A* search algorithm and applying it to the 8-puzzle problem to rearrange the blocks so that they are in order. It permitted to slide blocks horizontally or vertically into the blank square. The following shows a sequence of legal moves from an initial board position to the goal position.

```
    1  3        1     3        1  2  3        1  2  3        1  2  3
 4  2  5   =>   4  2  5   =>   4     5   =>   4  5     =>   4  5  6
 7  8  6        7  8  6        7  8  6        7  8  6        7  8

 initial                                                      goal
```

2. Code Summary

The Project consist of 4 classes

Program.cs – Main class from where execution begins, it contains the logic of calling other classes and functions.

AStarAlgo.cs – The main class from where all the logic computation happens. This class has all the different functions which include expanding neighbors, calculating heuristic distance, printing the matrices, etc.

Matrix.cs – Model class for storing initial, previous matrix, and the corresponding g value.

Node.cs – Model class for storing matrix as a node, it contains Priority Queue for maintaining the proper expansion order based on the f value. It also contains variables for storing heuristic, cost, and corresponding f value.

Logic:

The logic of the project is to store the initial matrix in a variable, expand its neighbors to store it in a priority queue with key as the corresponding f value (f value is calculated for every node, f = g + h). Based on the lowest key(f) value, the node is expanded till the final matrix is reached. The path from the goal state is then printed.

Note – We have used 0 for space when expanding the neighbors.

3. Language And IDE

- **Language:** C#
- **IDE:** Visual Studio 2022 Preview Edition (https://visualstudio.microsoft.com/vs/preview/)
- **Framework:** .Net 6.0 (https://dotnet.microsoft.com/download/dotnet/6.0)

Note: The implementation uses priority queue which is available **only** in .NET 6.0 and works in Visual Studio 2022 edition.

4. Global Variables

AstarAlgo.cs:

- InitialMatrix: Property to initialize Initial Matrix.
- Node: Property of type Node, Node will have the priority queue and the corresponding nodes g, f and h values.
- linkedListMatrix: A linkedlist to keep track of the correct order of path.
- Heuristic: Type of heuristic used, h1 for misplaced tiles and h2 for manhattan distance.
- LoopCount: LoopCount so that 8 puzzle doesn't run into infinite steps.

Matrix.cs:

- InitialMatrix: prop to store initial matrix
- PreviousMatrix: prop to store final matrix
- g: prop to store corresponding g value

Node.cs:

- int G: prop to store corresponding g value
- int F: prop to store corresponding F value
- int H: prop to store corresponding heuristic value

Program.cs

- initialMatrix: Initialized initial matrix to default value
- finalMatrix: Initialized final matrix to default value
- heuristic: variable for storing heuristic value, h1 for Misplaced Tiles, h2 for Manhattan Distance.

5. Functions/Procedures

AstarAlgo.cs:

- PrintMatrix: Method to print the matrix
- FindMatrixIndex: Method to find the matrix index based on it's values. Used to find index for Manhattan distance calculation.
- CalculateCost: Method for calculation heuristic distance h1 is Misplaced Tiles and h2 is Manhattan Distance.
- ComputeAstar: Method to implement the A* Algorithm. Recursive calls are there based on the least f values.
- Swap: Method to swap matrix elements, used in calculating child nodes.
- ExpandNeighbours: Method for calculating all possible combinations of moving the matrix, i.e all possible combinations of moving the tiles and storing them into the fringe.
- CompareMatrices: Method to compare the two matrices.
- PrintLinkedListMatrix: Method to print the LinkedList Matrix.

6. Program Structure

This section shows the screenshots of implementation in C#.

1. Program.cs: Takes input from user about initial state and goal state and calls the A* algorithm.

```csharp
using System;
using System.Collections.Generic;

namespace IntelligentSystem8_puzzleUsingAStar
{
    internal static class Program
    {
        // Program to demonstrate AStar
        static void Main(string[] args)
        {
            int[,] initialMatrix = { { 0, 1, 3 }, { 4, 2, 5 }, { 7, 8, 6 } }; // Initialized initial matrix to default value
            int[,] finalMatrix = { { 1, 2, 3 }, { 4, 5, 6, }, { 7, 8, 0 } }; // Initialized final matrix to default value

            string heuristic = "h2";

            Console.WriteLine("Enter the initial matrix in array format separated by spaces, eg  1 2 3 4 5 6 7 8 0, please consider 0 for space.");

            string str = Console.ReadLine();

            string[] strK = str.Split(" ");
            int countStr = 0;
            try
            {
                for (int i = 0; i < 3; i++)
                {
                    for (int j = 0; j < 3; j++)
                    {
                        initialMatrix[i, j] = Convert.ToInt32(strK[countStr]);
                        countStr++;
                    }
                }
                countStr = 0;
                Console.WriteLine("Enter the goal matrix in array format separated by spaces, eg  1 2 3 4 5 6 7 8 0, please consider 0 for space.");

                str=Console.ReadLine();

                strK = str.Split(" ");

                for (int i = 0; i < 3; i++)
                {
                    for (int j = 0; j < 3; j++)
                    {
                        finalMatrix[i, j] = Convert.ToInt32(strK[countStr]);
                        countStr++;
                    }
                }
            }
            catch (Exception e)
            {
                Console.WriteLine("Input not in correct format {0}",e);

                Environment.Exit(1);
            }

            Console.WriteLine("Choose 1 for Misplaced tiles and 2 for Manhattan Distance");

            var input = Console.ReadLine();

            switch(input)
            {
                case "1":
                    heuristic = "h1";
                   break;

                case "2":

                default:
                    heuristic = "h2";
                    break;
            }

            Matrix matrix = new Matrix(0, initialMatrix, initialMatrix);

            PriorityQueue<Matrix, int> priorityQueue = new PriorityQueue<Matrix, int>();

            priorityQueue.Enqueue(matrix, 0);

            Node node = new Node(0, 0, 0, priorityQueue);

            const int count = 1000;

            AStarAlgo aStarAlgo = new AStarAlgo(initialMatrix, node, heuristic,count);

            aStarAlgo.ComputeAstar(0, aStarAlgo.InitialMatrix, finalMatrix, node);

            AStarAlgo.PrintMatrix(initialMatrix);

            if (aStarAlgo.LoopCount <= 0)
            {
                Console.WriteLine("No Solution");
                Environment.Exit(0);
            }

            AStarAlgo.PrintLinkedListMatrix(aStarAlgo.linkedListMatrix);
        }
    }
}
```

2. Node.cs: Node class to store the node, a node will have corresponding f, g and h values.

```csharp
using System.Collections.Generic;

namespace IntelligentSystem8_puzzleUsingAStar
{
    // Node class to store the node, a node will have corresponding f g and h values.
    public class Node
    {
        #region Property
        public PriorityQueue<Matrix, int> PriorityQueue { get; set; } // prop for priority queue to maintain original expansion order.
        public int G { get; set; } // prop to store corresponding g value
        public int F { get; set; } // prop to store corresponding f value

        public int H { get; set; } // prop to store corresponding h value
        #endregion

        #region Constructor
        public Node(int g, int f, int h, PriorityQueue<Matrix, int> priorityQueue)
        {
            this.G = g;
            this.F = f;
            this.H = h;
            this.PriorityQueue = priorityQueue;
        }
        #endregion
    }
}

```

3. Matrix.cs: A matrix class having initial and previous matrix and corresponding g values.

```csharp
namespace IntelligentSystem8_puzzleUsingAStar
{
    // A matrix class having initial and previous matrix and corresponding g values.
    public class Matrix
    {
        #region Properties
        public int[,] InitialMatrix { get; set; } // prop to store initial matrix

        public int[,] PreviousMatrix { get; set; } // prop to store final matrix
        public int g { get; set; } // prop to store corresponding g value
        #endregion

        #region Constructor
        public Matrix(int g, int[,] InitialMatrix, int[,] PreviousMatrix)
        {
            this.g = g;
            this.InitialMatrix = InitialMatrix;
            this.PreviousMatrix = PreviousMatrix;
        }
        #endregion
    }
}

```

4. AstarAlgo.cs:

```
using System;
using System.Collections.Generic;

namespace IntelligentSystem8_puzzleUsingAStar
{
    public class AStarAlgo
    {
        #region Properties
        public int[,] InitialMatrix { get; set; } // Property to initialize Initial Matrix.

        public Node Node { get; set; } // Property of type Node, Node will have the priority queue and the corresponding nodes g,f and h values.

        public LinkedList<int[,]> linkedListMatrix; // A linkedlist to keep track of the correct order of path.

        public string Heuristic { get; set; } // Type of heuristic used, h1 for misplaced tiles and h2 for manhattan distance.

        public int LoopCount { get; set; } // LoopCount so that 8 puzzle doesn't run into infinite steps.

        #endregion

        #region Constructor
        public AStarAlgo(int[,] InitialMatrix, Node Node, string Heuristic, int LoopCount)
        {
            this.InitialMatrix = InitialMatrix;
            this.Node = Node;
            linkedListMatrix = new LinkedList<int[,]>();
            this.Heuristic = Heuristic;
            this.LoopCount = LoopCount;
        }

        #endregion

        #region Public Methods
        // Method to print the matrix
        public static void PrintMatrix(int[,] matrix)
        {
            for (int i = 0; i < 3; i++)
            {
                for (int j = 0; j < 3; j++)
                {
                    Console.Write(" {0}", matrix[i, j]);
                }
                Console.WriteLine();
            }
            Console.WriteLine("*********");
        }
```

```
// Method to find the matrix index based on it's values. Used to find index for Manhattan distance calculation.
        public static (int, int) FindMatrixIndex(int[,] finalMatrix, int element)
        {
            for (int i = 0; i < 3; i++)
            {
                for (int j = 0; j < 3; j++)
                {
                    if (element == finalMatrix[i, j])
                        return new(i, j);
                }
            }

            return default;
        }

        // Method for calculation heuristic distance h1 is Misplaced Tiles and h2 is Manhattan Distance.
        public int CalculateCost(int[,] tempMatrix, int[,] finalMatrix)
        {
            int dist = 0;
            switch (Heuristic)
            {
                case "h1":
                    for (int i = 0; i < 3; i++)
                    {
                        for (int j = 0; j < 3; j++)
                        {
                            if (tempMatrix[i, j] != finalMatrix[i, j] && tempMatrix[i, j] != 0)
                                dist++;
                        }
                    }

                    return dist;

                case "h2":
                    {
                        for (int i = 0; i < 3; i++)
                        {
                            for (int j = 0; j < 3; j++)
                            {
                                int x;
                                int y;
                                (x, y) = FindMatrixIndex(finalMatrix, tempMatrix[i, j]);

                                dist += Math.Abs(x - i) + Math.Abs(y - j);
                            }
                        }

                        return dist;
                    }
            }

            return 0;
        }

        // Method to implement the A* Algorithm. Recursive calls are there based on the least f values.
        public void ComputeAstar(int g, int[,] initialMatrix, int[,] finalMatrix, Node node)
        {
            LoopCount--;

            if (!CompareMatrices(initialMatrix, finalMatrix) && LoopCount > 0)
            {
                for (int i = 0; i < 3; i++)
                {
                    for (int j = 0; j < 3; j++)
                    {
                        if (initialMatrix[i, j] == 0)
                        {
                            ExpandNeighbours(initialMatrix, i, j, g, finalMatrix, node.PriorityQueue.Peek().PreviousMatrix);

                            if (node.PriorityQueue.Peek().InitialMatrix == initialMatrix)
                                node.PriorityQueue.Dequeue();

                            linkedListMatrix.AddLast(node.PriorityQueue.Peek().InitialMatrix);

                            var initialTempMatrix = node.PriorityQueue.Peek().InitialMatrix;

                            node.PriorityQueue.Dequeue();

                            ComputeAstar(node.PriorityQueue.Peek().g + 1, initialTempMatrix, finalMatrix, node);
                        }
                    }
                }
            }
        }

        // Method to swap matrix elements, used in calculating child nodes.
        public static void Swap(int i, int j, int x, int y, int[,] tempMatrix)
        {
            int temp = tempMatrix[i, j];
            tempMatrix[i, j] = tempMatrix[x, y];
            tempMatrix[x, y] = temp;
        }
```

```
// Method for calculating all possible combinations of moving the matrix, i.e all possible combinations of moving the tiles and storing them into the fringe.
        public void ExpandNeighbours(int[,] initialMatrix, int i, int j, int g, int[,] finalMatrix, int[,] previousMatrix)
        {
            int h, f;
            int[,] tempMatrix = (int[,])initialMatrix.Clone();

            var matrix = new Matrix(g, tempMatrix, previousMatrix);

            //var node = new Node(g, f, h);

            if (i == 0)
            {
                if (j == 0)
                {
                    Swap(i, j, 0, 1, tempMatrix);
                    matrix = new Matrix(g, tempMatrix, previousMatrix);
                    h = CalculateCost(tempMatrix, finalMatrix);
                    f = g + h;
                    matrix.g = g;
                    Node.PriorityQueue.Enqueue(matrix, f);

                    tempMatrix = (int[,])initialMatrix.Clone();
                    Swap(i, j, 1, 0, tempMatrix);
                    matrix = new Matrix(g, tempMatrix, previousMatrix);
                    h = CalculateCost(tempMatrix, finalMatrix);
                    f = g + h;
                    matrix.g = g;
                    Node.PriorityQueue.Enqueue(matrix, f);
                }
                else if (j == 1)
                {
                    Swap(i, j, 0, 2, tempMatrix);
                    matrix = new Matrix(g, tempMatrix, previousMatrix);
                    h = CalculateCost(tempMatrix, finalMatrix);
                    f = g + h;
                    matrix.g = g;
                    Node.PriorityQueue.Enqueue(matrix, f);

                    tempMatrix = (int[,])initialMatrix.Clone();
                    Swap(i, j, 1, 1, tempMatrix);
                    matrix = new Matrix(g, tempMatrix, previousMatrix);
                    h = CalculateCost(tempMatrix, finalMatrix);
                    f = g + h;
                    matrix.g = g;
                    Node.PriorityQueue.Enqueue(matrix, f);

                    tempMatrix = (int[,])initialMatrix.Clone();
                    Swap(i, j, 0, 0, tempMatrix);
                    matrix = new Matrix(g, tempMatrix, previousMatrix);
                    h = CalculateCost(tempMatrix, finalMatrix);
                    f = g + h;
                    matrix.g = g;
                    Node.PriorityQueue.Enqueue(matrix, f);
                }
                else
                {
                    Swap(i, j, 0, 1, tempMatrix);
                    matrix = new Matrix(g, tempMatrix, previousMatrix);
                    h = CalculateCost(tempMatrix, finalMatrix);
                    f = g + h;
                    matrix.g = g;
                    Node.PriorityQueue.Enqueue(matrix, f);

                    tempMatrix = (int[,])initialMatrix.Clone();
                    Swap(i, j, 1, 2, tempMatrix);
                    matrix = new Matrix(g, tempMatrix, previousMatrix);
                    h = CalculateCost(tempMatrix, finalMatrix);
                    f = g + h;
                    matrix.g = g;
                    Node.PriorityQueue.Enqueue(matrix, f);
                }
            }
```

```
else if (i == 1)
            {
                if (j == 0)
                {
                    Swap(i, j, 0, 0, tempMatrix);
                    matrix = new Matrix(g, tempMatrix, previousMatrix);
                    h = CalculateCost(tempMatrix, finalMatrix);
                    f = g + h;
                    matrix.g = g;
                    Node.PriorityQueue.Enqueue(matrix, f);

                    tempMatrix = (int[,])initialMatrix.Clone();
                    Swap(i, j, 1, 1, tempMatrix);
                    matrix = new Matrix(g, tempMatrix, previousMatrix);
                    h = CalculateCost(tempMatrix, finalMatrix);
                    f = g + h;
                    matrix.g = g;
                    Node.PriorityQueue.Enqueue(matrix, f);

                    tempMatrix = (int[,])initialMatrix.Clone();
                    Swap(i, j, 2, 0, tempMatrix);
                    matrix = new Matrix(g, tempMatrix, previousMatrix);
                    h = CalculateCost(tempMatrix, finalMatrix);
                    f = g + h;
                    matrix.g = g;
                    Node.PriorityQueue.Enqueue(matrix, f);
                }
                else if (j == 1)
                {
                    Swap(i, j, 0, 1, tempMatrix);
                    matrix = new Matrix(g, tempMatrix, previousMatrix);
                    h = CalculateCost(tempMatrix, finalMatrix);
                    f = g + h;
                    matrix.g = g;
                    Node.PriorityQueue.Enqueue(matrix, f);

                    tempMatrix = (int[,])initialMatrix.Clone();
                    Swap(i, j, 1, 0, tempMatrix);
                    matrix = new Matrix(g, tempMatrix, previousMatrix);
                    h = CalculateCost(tempMatrix, finalMatrix);
                    f = g + h;
                    matrix.g = g;
                    Node.PriorityQueue.Enqueue(matrix, f);

                    tempMatrix = (int[,])initialMatrix.Clone();
                    Swap(i, j, 2, 1, tempMatrix);
                    matrix = new Matrix(g, tempMatrix, previousMatrix);
                    h = CalculateCost(tempMatrix, finalMatrix);
                    f = g + h;
                    matrix.g = g;
                    Node.PriorityQueue.Enqueue(matrix, f);

                    tempMatrix = (int[,])initialMatrix.Clone();
                    Swap(i, j, 1, 2, tempMatrix);
                    matrix = new Matrix(g, tempMatrix, previousMatrix);
                    h = CalculateCost(tempMatrix, finalMatrix);
                    f = g + h;
                    matrix.g = g;
                    Node.PriorityQueue.Enqueue(matrix, f);
                }
                else
                {
                    Swap(i, j, 0, 2, tempMatrix);
                    matrix = new Matrix(g, tempMatrix, previousMatrix);
                    h = CalculateCost(tempMatrix, finalMatrix);
                    f = g + h;
                    matrix.g = g;
                    Node.PriorityQueue.Enqueue(matrix, f);

                    tempMatrix = (int[,])initialMatrix.Clone();
                    Swap(i, j, 1, 1, tempMatrix);
                    matrix = new Matrix(g, tempMatrix, previousMatrix);
                    h = CalculateCost(tempMatrix, finalMatrix);
                    f = g + h;
                    matrix.g = g;
                    Node.PriorityQueue.Enqueue(matrix, f);

                    tempMatrix = (int[,])initialMatrix.Clone();
                    Swap(i, j, 2, 2, tempMatrix);
                    matrix = new Matrix(g, tempMatrix, previousMatrix);
                    h = CalculateCost(tempMatrix, finalMatrix);
                    f = g + h;
                    matrix.g = g;
                    Node.PriorityQueue.Enqueue(matrix, f);
                }
            }
```

```
else
        {
            if (j == 0)
            {
                Swap(i, j, 1, 0, tempMatrix);
                matrix = new Matrix(g, tempMatrix, previousMatrix);
                h = CalculateCost(tempMatrix, finalMatrix);
                f = g + h;
                matrix.g = g;
                Node.PriorityQueue.Enqueue(matrix, f);

                tempMatrix = (int[,])initialMatrix.Clone();
                Swap(i, j, 2, 1, tempMatrix);
                matrix = new Matrix(g, tempMatrix, previousMatrix);
                h = CalculateCost(tempMatrix, finalMatrix);
                f = g + h;
                matrix.g = g;
                Node.PriorityQueue.Enqueue(matrix, f);
            }
            else if (j == 1)
            {
                Swap(i, j, 1, 1, tempMatrix);
                matrix = new Matrix(g, tempMatrix, previousMatrix);
                h = CalculateCost(tempMatrix, finalMatrix);
                f = g + h;
                matrix.g = g;
                Node.PriorityQueue.Enqueue(matrix, f);

                tempMatrix = (int[,])initialMatrix.Clone();
                Swap(i, j, 2, 0, tempMatrix);
                matrix = new Matrix(g, tempMatrix, previousMatrix);
                h = CalculateCost(tempMatrix, finalMatrix);
                f = g + h;
                matrix.g = g;
                Node.PriorityQueue.Enqueue(matrix, f);

                tempMatrix = (int[,])initialMatrix.Clone();
                Swap(i, j, 2, 2, tempMatrix);
                matrix = new Matrix(g, tempMatrix, previousMatrix);
                h = CalculateCost(tempMatrix, finalMatrix);
                f = g + h;
                matrix.g = g;
                Node.PriorityQueue.Enqueue(matrix, f);
            }
            else
            {
                Swap(i, j, 1, 2, tempMatrix);
                matrix = new Matrix(g, tempMatrix, previousMatrix);
                h = CalculateCost(tempMatrix, finalMatrix);
                f = g + h;
                matrix.g = g;
                Node.PriorityQueue.Enqueue(matrix, f);

                tempMatrix = (int[,])initialMatrix.Clone();
                Swap(i, j, 2, 1, tempMatrix);
                matrix = new Matrix(g, tempMatrix, previousMatrix);
                h = CalculateCost(tempMatrix, finalMatrix);
                f = g + h;
                matrix.g = g;
                Node.PriorityQueue.Enqueue(matrix, f);
            }
        }
    }
```

```
// Method to compare the two matrices.
        public static bool CompareMatrices(int[,] initialMatrix, int[,] finalMatrix)
        {
            for (int i = 0; i < 3; i++)
            {
                for (int j = 0; j < 3; j++)
                {
                    if (initialMatrix[i, j] != finalMatrix[i, j])
                    {
                        return false;
                    }
                }
            }
            return true;
        }

        // Method to print the LinkedList Matrix.
        public static void PrintLinkedListMatrix(LinkedList<int[,]> linkedListMatrix)
        {
            foreach (var item in linkedListMatrix)
            {
                for (int i = 0; i < 3; i++)
                {
                    for (int j = 0; j < 3; j++)
                    {
                        Console.Write(" {0}", item[i, j]);
                    }
                    Console.WriteLine();
                }
                Console.WriteLine("*********");
            }
        }
        #endregion
    }
}
```

7 Analysis of 6 Input / Output Cases

Following subsections have the analysis the 8-puzzle problem using different inputs and outputs. Each case is solved using Misplaced Tiles and Manhattan distance.

Each case shows the leaves generated and its heuristic after exploration of node, and shows the best path later.

Case 1

Initial:

2 8 1
3 4 6
7 5 0

Goal:

3 2 1
8 0 4
7 5 6

1.1 Misplaced tiles:

Initial:

2 8 1
3 4 6
7 5 0

Goal:

3 2 1
8 0 4
7 5 6

Heuristic: Misplaced Tiles

Leaves:

2 8 1
3 4 0
7 5 6
f(n)=5, g(n)= 1, h(n)= 4

2 8 1
3 4 6
7 0 5
f(n)=7, g(n)= 1, h(n)= 6

Leaves:

2 8 0
3 4 1
7 5 6
f(n)=7, g(n)= 2, h(n)= 5

2 8 1
3 0 4
7 5 6
f(n)=5, g(n)= 2, h(n)= 3

2 8 1
3 4 6
7 5 0
Visited

Leaves:

2 0 1
3 8 4
7 5 6
f(n)=6, g(n)= 3, h(n)= 3

2 8 1
0 3 4
7 5 6
f(n)=6, g(n)= 3, h(n)= 3

2 8 1
3 4 0
7 5 6
Visited

2 8 1
3 5 4
7 0 6
f(n)=7, g(n)= 3, h(n)= 4

Leaves:

0 2 1
3 8 4
7 5 6
f(n)=6, g(n)= 4, h(n)= 2

2 1 0
3 8 4
7 5 6
f(n)=8, g(n)= 4, h(n)= 4

2 8 1
3 0 4
7 5 6
Visited

Leaves:

0 8 1
2 3 4
7 5 6
f(n)=7, g(n)= 4, h(n)= 3

2 8 1
3 0 4
7 5 6
Visited

2 8 1
7 3 4
0 5 6
f(n)=8, g(n)= 4, h(n)= 4

Leaves:

2 0 1
3 8 4
7 5 6
Visited

3 2 1
0 8 4
7 5 6
f(n)=6, g(n)= 5, h(n)= 1

Leaves:

0 2 1
3 8 4
7 5 6
Visited

3 2 1
8 0 4
7 5 6
f(n)=6, g(n)= 6, h(n)= 0

3 2 1
7 8 4
0 5 6
f(n)=8, g(n)= 6, h(n)= 2

Path:

2 8 1
3 4 6
7 5 0

2 8 1
3 4 0
7 5 6

2 8 1
3 0 4
7 5 6

2 0 1
3 8 4
7 5 6

0 2 1
3 8 4
7 5 6

3 2 1
0 8 4
7 5 6

3 2 1
8 0 4
7 5 6

Generated Nodes = 15
Explored Nodes = 7
Total moves = 6

1.2 Manhattan Distance:

Initial:

2 8 1
3 4 6
7 5 0

Goal:

3 2 1
8 0 4
7 5 6

Heuristic: Manhattan Distance

Leaves:

2 8 1
3 4 0
7 5 6
f(n)=6, g(n)= 1, h(n)= 5

2 8 1
3 4 6
7 0 5
f(n)=8, g(n)= 1, h(n)= 7

Leaves:

2 8 0
3 4 1
7 5 6
f(n)=8, g(n)= 2, h(n)= 6

2 8 1
3 0 4
7 5 6
f(n)=6, g(n)= 2, h(n)= 4

2 8 1
3 4 6
7 5 0
Visited

Leaves:

2 0 1
3 8 4
7 5 6
f(n)=6, g(n)= 3, h(n)= 3

2 8 1
0 3 4
7 5 6
f(n)=8, g(n)= 3, h(n)= 5

2 8 1
3 4 0
7 5 6
Visited

2 8 1
3 5 4
7 0 6
f(n)=8, g(n)= 3, h(n)= 5

Leaves:
0 2 1
3 8 4
7 5 6
f(n)=6, g(n)= 4, h(n)= 2

2 1 0
3 8 4

7 5 6
f(n)=8, g(n)= 4, h(n)= 4

2 8 1
3 0 4
7 5 6
Visited

Leaves:

2 0 1
3 8 4
7 5 6
Visited

3 2 1
0 8 4
7 5 6
f(n)=6, g(n)= 5, h(n)= 1

Leaves:

0 2 1
3 8 4
7 5 6
Visited

3 2 1
8 0 4
7 5 6
f(n)=6, g(n)= 6, h(n)= 0

3 2 1
7 8 4
0 5 6
f(n)=8, g(n)= 6, h(n)= 2

Path:

2 8 1
3 4 6
7 5 0

2 8 1
3 4 0
7 5 6

2 8 1
3 0 4
7 5 6

2 0 1
3 8 4
7 5 6

0 2 1
3 8 4
7 5 6

3 2 1
0 8 4
7 5 6

3 2 1
8 0 4
7 5 6

Generated Nodes = 13
Explored Nodes = 6
Total moves = 6

Case 2

Initial:

1 2 3
5 6 0
7 8 4

Goal:

1 2 3
5 8 6
0 7 4

2.1 Misplaced Tiles

Initial:

1 2 3
5 6 0
7 8 4

Goal:

1 2 3
5 8 6
0 7 4

Heuristic: Misplaced Tiles

Leaves:

1 2 0
5 6 3
7 8 4

f(n)=5, g(n)= 1, h(n)= 4

1 2 3
5 0 6
7 8 4
f(n)=3, g(n)= 1, h(n)= 2

1 2 3
5 6 4
7 8 0
f(n)=5, g(n)= 1, h(n)= 4

Leaves:

1 0 3
5 2 6
7 8 4
f(n)=5, g(n)= 2, h(n)= 3

1 2 3
0 5 6
7 8 4
f(n)=5, g(n)= 2, h(n)= 3

1 2 3
5 6 0
7 8 4
Visited

1 2 3
5 8 6
7 0 4
f(n)=3, g(n)= 2, h(n)= 1

Leaves:

1 2 3
5 0 6
7 8 4
Visited

1 2 3
5 8 6
0 7 4
f(n)=3, g(n)= 3, h(n)= 0

1 2 3
5 8 6
7 4 0
f(n)=5, g(n)= 3, h(n)= 2

Path:

1 2 3
5 6 0
7 8 4

1 2 3
5 0 6
7 8 4

1 2 3
5 8 6
7 0 4

1 2 3
5 8 6
0 7 4

Generated Nodes = 9
Explored Nodes = 5
Total moves = 3

2.2 Manhattan Distance:

Initial:

1 2 3
5 6 0
7 8 4

Goal:

1 2 3
5 8 6
0 7 4

Heuristic: Manhattan Distance

Leaves:

1 2 0
5 6 3
7 8 4
f(n)=5, g(n)= 1, h(n)= 4

1 2 3
5 0 6
7 8 4
f(n)=3, g(n)= 1, h(n)= 2

1 2 3
5 6 4
7 8 0
f(n)=5, g(n)= 1, h(n)= 4

Leaves:

1 0 3
5 2 6
7 8 4
f(n)=5, g(n)= 2, h(n)= 3

1 2 3
0 5 6
7 8 4
f(n)=5, g(n)= 2, h(n)= 3

1 2 3
5 6 0
7 8 4
Visited

1 2 3
5 8 6
7 0 4
f(n)=3, g(n)= 2, h(n)= 1

Leaves:

1 2 3
5 0 6
7 8 4
Visited

1 2 3
5 8 6
0 7 4
f(n)=3, g(n)= 3, h(n)= 0

1 2 3
5 8 6
7 4 0
f(n)=5, g(n)= 3, h(n)= 2

Path:

1 2 3
5 6 0
7 8 4

1 2 3
5 0 6
7 8 4

1 2 3
5 8 6
7 0 4

1 2 3
5 8 6
0 7 4

Generated Nodes = 9
Explored Nodes = 5
Total moves = 3

Case 3

3.1 Misplaced Tiles:

Initial:

0 1 3
4 2 5
7 8 6

Goal:

123
456
780

Heuristic: Misplaced Tiles

Leaves:

1 0 3
4 2 5
7 8 6
f(n)=4, g(n)= 1, h(n)= 3

4 1 3
0 2 5
7 8 6
f(n)=6, g(n)= 1, h(n)= 5

Leaves:

0 1 3
4 2 5
7 8 6
Visited

1 3 0
4 2 5
7 8 6
f(n)=6, g(n)= 2, h(n)= 4

1 2 3
4 0 5
7 8 6
f(n)=4, g(n)= 2, h(n)= 2

Leaves:

1 0 3
4 2 5
7 8 6
Visited

1 2 3
0 4 5
7 8 6
f(n)=6, g(n)= 3, h(n)= 3

1 2 3
4 5 0
7 8 6
f(n)=4, g(n)= 3, h(n)= 1

1 2 3
4 8 5
7 0 6
f(n)=6, g(n)= 3, h(n)= 3

Leaves:

1 2 0
4 5 3
7 8 6
f(n)=6, g(n)= 4, h(n)= 2

1 2 3
4 0 5
7 8 6
Visited

1 2 3
4 5 6
7 8 0
f(n)=4, g(n)= 4, h(n)= 0

Path:

0 1 3
4 2 5
7 8 6

1 0 3
4 2 5
7 8 6

1 2 3
4 0 5
7 8 6

1 2 3
4 5 0
7 8 6

1 2 3
4 5 6
7 8 0

Generated Nodes = 10
Explored Nodes = 5
Total moves = 4

3. 2 Manhattan Distance:

Initial:

013
425
786

Goal:

123
456
780

Heuristic: Manhattan Distance

Leaves:

1 0 3
4 2 5
7 8 6
f(n)=4, g(n)= 1, h(n)= 3

4 1 3
0 2 5
7 8 6
f(n)=6, g(n)= 1, h(n)= 5

Leaves:

0 1 3
4 2 5
7 8 6
Visited

1 3 0
4 2 5
7 8 6
f(n)=6, g(n)= 2, h(n)= 4

1 2 3
4 0 5
7 8 6
f(n)=4, g(n)= 2, h(n)= 2

Leaves:

1 0 3
4 2 5
7 8 6
Visited

1 2 3
0 4 5
7 8 6
f(n)=6, g(n)= 3, h(n)= 3

1 2 3
4 5 0
7 8 6
f(n)=4, g(n)= 3, h(n)= 1

1 2 3
4 8 5
7 0 6
f(n)=6, g(n)= 3, h(n)= 3

Leaves:

1 2 0
4 5 3
7 8 6
f(n)=6, g(n)= 4, h(n)= 2
1 2 3

4 0 5
7 8 6
Visited

1 2 3
4 5 6
7 8 0
f(n)=4, g(n)= 4, h(n)= 0

Path:

0 1 3
4 2 5
7 8 6

1 0 3
4 2 5
7 8 6

1 2 3
4 0 5
7 8 6

1 2 3
4 5 0
7 8 6

1 2 3
4 5 6
7 8 0

Generated Nodes = 10
Explored Nodes = 5
Total moves = 4

Case 4

4.1 Misplaced Tiles:

Initial:

2 8 3
1 6 4
7 0 5

Goal:

1 2 3
8 0 4
7 6 5

Heuristic: Misplaced Tiles

Leaves:

2 8 3
1 0 4
7 6 5
f(n)=4, g(n)= 1, h(n)= 3

2 8 3
1 6 4
0 7 5
f(n)=6, g(n)= 1, h(n)= 5

2 8 3
1 6 4
7 5 0
f(n)=6, g(n)= 1, h(n)= 5

Leaves:

2 0 3
1 8 4
7 6 5
f(n)=5, g(n)= 2, h(n)= 3

2 8 3
0 1 4
7 6 5
f(n)=5, g(n)= 2, h(n)= 3

2 8 3
1 4 0
7 6 5
f(n)=6, g(n)= 2, h(n)= 4

2 8 3
1 6 4
7 0 5
Visited

Leaves:

0 2 3
1 8 4
7 6 5
f(n)=5, g(n)= 3, h(n)= 2

2 3 0
1 8 4
7 6 5
f(n)=7, g(n)= 3, h(n)= 4

2 8 3
1 0 4
7 6 5
Visited

Leaves:

0 8 3
2 1 4
7 6 5
f(n)=6, g(n)= 3, h(n)= 3

2 8 3
1 0 4
7 6 5
Visited

2 8 3
7 1 4
0 6 5
f(n)=7, g(n)= 3, h(n)= 4

Leaves:

2 0 3
1 8 4
7 6 5
Visited

1 2 3
0 8 4
7 6 5
f(n)=5, g(n)= 4, h(n)= 1

Leaves:

0 2 3
1 8 4
7 6 5
Visited
1 2 3
8 0 4
7 6 5
f(n)=5, g(n)= 5, h(n)= 0

1 2 3
7 8 4
0 6 5
f(n)=7, g(n)= 5, h(n)= 2

Path:

2 8 3
1 6 4
7 0 5

2 8 3
1 0 4
7 6 5

2 0 3
1 8 4
7 6 5

0 2 3
1 8 4
7 6 5

1 2 3
0 8 4
7 6 5

1 2 3
8 0 4
7 6 5

Generated Nodes = 14
Explored Nodes = 7
Total moves = 5

4.2 Manhattan Distance:

Leaves:

2 8 3
1 0 4
7 6 5
f(n)=5, g(n)= 1, h(n)= 4

2 8 3
1 6 4
0 7 5
f(n)=7, g(n)= 1, h(n)= 6

2 8 3
1 6 4
7 5 0
f(n)=7, g(n)= 1, h(n)= 6

Leaves:

2 0 3
1 8 4
7 6 5
f(n)=5, g(n)= 2, h(n)= 3

2 8 3
0 1 4
7 6 5
f(n)=7, g(n)= 2, h(n)= 5

2 8 3
1 4 0
7 6 5
f(n)=7, g(n)= 2, h(n)= 5

2 8 3
1 6 4
7 0 5
Visited

Leaves:

0 2 3
1 8 4
7 6 5
f(n)=5, g(n)= 3, h(n)= 2

2 3 0
1 8 4
7 6 5
f(n)=7, g(n)= 3, h(n)= 4

2 8 3
1 0 4
7 6 5
Visited

Leaves:

2 0 3
1 8 4
7 6 5
Visited

1 2 3
0 8 4
7 6 5
f(n)=5, g(n)= 4, h(n)= 1

Leaves:

0 2 3
1 8 4
7 6 5
Visited

1 2 3
8 0 4
7 6 5
f(n)=5, g(n)= 5, h(n)= 0

1 2 3
7 8 4
0 6 5
f(n)=7, g(n)= 5, h(n)= 2

Path:

2 8 3
1 6 4
7 0 5

2 8 3
1 0 4
7 6 5

2 0 3
1 8 4
7 6 5

0 2 3
1 8 4
7 6 5

1 2 3
0 8 4
7 6 5

1 2 3
8 0 4
7 6 5

Generated Nodes = 12
Explored Nodes = 6
Total moves = 5

Case 5

Initial:

1 8 2
0 4 3
7 6 5

Goal:

1 2 3
4 5 6
7 8 0

5.1 Misplaced tiles:

Initial:

1 8 2
0 4 3
7 6 5

Goal:

1 2 3
4 5 6
7 8 0

Heuristic: Misplaced Tiles

Leaves:

0 8 2
1 4 3
7 6 5
f(n)=8, g(n)= 1, h(n)= 7

1 8 2
4 0 3
7 6 5
f(n)=6, g(n)= 1, h(n)= 5

1 8 2
7 4 3
0 6 5
f(n)=8, g(n)= 1, h(n)= 7

Leaves:

1 0 2
4 8 3
7 6 5
f(n)=7, g(n)= 2, h(n)= 5

1 8 2
0 4 3
7 6 5
Visited

1 8 2
4 3 0
7 6 5
f(n)=7, g(n)= 2, h(n)= 5

1 8 2
4 6 3
7 0 5
f(n)=7, g(n)= 2, h(n)= 5

Leaves:

0 1 2
4 8 3
7 6 5
f(n)=9, g(n)= 3, h(n)= 6

1 2 0
4 8 3
7 6 5
f(n)=7, g(n)= 3, h(n)= 4

1 8 2
4 0 3
7 6 5
Visited

Leaves:

1 8 0
4 3 2
7 6 5
f(n)=8, g(n)= 3, h(n)= 5

1 8 2
4 0 3
7 6 5
Visited

1 8 2
4 3 5
7 6 0
f(n)=8, g(n)= 3, h(n)= 5

Leaves:

1 8 2
4 0 3
7 6 5
Visited

1 8 2
4 6 3
0 7 5
f(n)=9, g(n)= 3, h(n)= 6

1 8 2
4 6 3
7 5 0
f(n)=8, g(n)= 3, h(n)= 5

Leaves:

1 0 2
4 8 3
7 6 5
Visited

1 2 3
4 8 0
7 6 5
f(n)=7, g(n)= 4, h(n)= 3

Leaves:

1 2 0
4 8 3
7 6 5
Visited

1 2 3
4 0 8
7 6 5
f(n)=8, g(n)= 5, h(n)= 3

1 2 3
4 8 5
7 6 0
f(n)=8, g(n)= 5, h(n)= 3

Leaves:

8 0 2
1 4 3
7 6 5
f(n)=9, g(n)= 2, h(n)= 7

1 8 2
0 4 3
7 6 5
Visited

Leaves:

1 8 2
0 4 3
7 6 5
Visited

1 8 2
7 4 3
6 0 5
f(n)=9, g(n)= 2, h(n)= 7

Leaves:

1 0 8
4 3 2
7 6 5
f(n)=9, g(n)= 4, h(n)= 5

1 8 2
4 3 0

7 6 5
Visited

Leaves:

1 8 2
4 3 0
7 6 5
Visited

1 8 2
4 3 5
7 0 6
f(n)=9, g(n)= 4, h(n)= 5

Leaves:

1 8 2
4 6 0
7 5 3
f(n)=9, g(n)= 4, h(n)= 5

1 8 2
4 6 3
7 0 5
Visited

Leaves:

1 0 3
4 2 8
7 6 5
f(n)=10, g(n)= 6, h(n)= 4

1 2 3
0 4 8
7 6 5
f(n)=10, g(n)= 6, h(n)= 4

1 2 3
4 8 0
7 6 5
Visited

1 2 3
4 6 8
7 0 5
f(n)=9, g(n)= 6, h(n)= 3

Leaves:

1 2 3
4 8 0
7 6 5
Visited

1 2 3
4 8 5
7 0 6
f(n)=9, g(n)= 6, h(n)= 3

Leaves:

1 0 2
4 8 3
7 6 5
Visited

4 1 2
0 8 3
7 6 5
f(n)=11, g(n)= 4, h(n)= 7

Leaves:

1 8 2
0 6 3
4 7 5
f(n)=11, g(n)= 4, h(n)= 7

1 8 2
4 6 3
7 0 5
Visited

Leaves:

0 8 2
1 4 3
7 6 5
Visited

8 2 0
1 4 3
7 6 5
f(n)=9, g(n)= 3, h(n)= 6

8 4 2
1 0 3
7 6 5
f(n)=10, g(n)= 3, h(n)= 7

Leaves:

1 8 2
7 0 3
6 4 5
f(n)=10, g(n)= 3, h(n)= 7

1 8 2
7 4 3
0 6 5
Visited

1 8 2
7 4 3
6 5 0
f(n)=10, g(n)= 3, h(n)= 7

Leaves:

0 1 8
4 3 2
7 6 5
f(n)=11, g(n)= 5, h(n)= 6

1 8 0
4 3 2
7 6 5
Visited

1 3 8
4 0 2
7 6 5
f(n)=10, g(n)= 5, h(n)= 5

Leaves:

1 8 2

4 0 5
7 3 6
f(n)=10, g(n)= 5, h(n)= 5

1 8 2
4 3 5
0 7 6
f(n)=11, g(n)= 5, h(n)= 6

1 8 2
4 3 5
7 6 0
Visited

Leaves:

1 8 0
4 6 2
7 5 3
f(n)=10, g(n)= 5, h(n)= 5

1 8 2
4 0 6
7 5 3
f(n)=9, g(n)= 5, h(n)= 4

1 8 2
4 6 3
7 5 0
Visited

Leaves:

1 2 3
4 0 8
7 6 5
Visited

1 2 3
4 6 8
0 7 5
f(n)=11, g(n)= 7, h(n)= 4

1 2 3
4 6 8
7 5 0
f(n)=10, g(n)= 7, h(n)= 3

Leaves:

1 2 3
4 0 5

7 8 6
f(n)=9, g(n)= 7, h(n)= 2

1 2 3
4 8 5
0 7 6
f(n)=11, g(n)= 7, h(n)= 4

1 2 3
4 8 5
7 6 0
Visited

Leaves:

8 0 2
1 4 3
7 6 5
Visited

8 2 3
1 4 0
7 6 5
f(n)=9, g(n)= 4, h(n)= 5

Leaves:

1 0 2
4 8 6
7 5 3
f(n)=10, g(n)= 6, h(n)= 4

1 8 2
0 4 6
7 5 3
f(n)=11, g(n)= 6, h(n)= 5

1 8 2
4 6 0
7 5 3
Visited

1 8 2
4 5 6
7 0 3
f(n)=9, g(n)= 6, h(n)= 3

Leaves:

1 0 3
4 2 5
7 8 6

f(n)=11, g(n)= 8, h(n)= 3

1 2 3
0 4 5
7 8 6
f(n)=11, g(n)= 8, h(n)= 3

1 2 3
4 5 0
7 8 6
f(n)=9, g(n)= 8, h(n)= 1

1 2 3
4 8 5
7 0 6
Visited

Leaves:

8 2 0
1 4 3
7 6 5
Visited

8 2 3
1 0 4
7 6 5
f(n)=10, g(n)= 5, h(n)= 5

8 2 3
1 4 5
7 6 0
f(n)=10, g(n)= 5, h(n)= 5

Leaves:

1 8 2
4 0 6
7 5 3
Visited

1 8 2
4 5 6
0 7 3
f(n)=11, g(n)= 7, h(n)= 4
1 8 2
4 5 6
7 3 0
f(n)=10, g(n)= 7, h(n)= 3

Leaves:

1 2 0
4 5 3
7 8 6
f(n)=11, g(n)= 9, h(n)= 2

1 2 3
4 0 5
7 8 6
Visited

1 2 3
4 5 6
7 8 0
f(n)=9, g(n)= 9, h(n)= 0

Path:

1 8 2
0 4 3
7 6 5

1 8 2
4 0 3
7 6 5

1 0 2
4 8 3
7 6 5

1 2 0
4 8 3
7 6 5

1 2 3
4 8 0
7 6 5

1 2 3
4 8 5
7 6 0

1 2 3
4 8 5
7 0 6

1 2 3
4 0 5

7 8 6

1 2 3
4 5 0
7 8 6

1 2 3
4 5 6
7 8 0

Generated Nodes = 54
Explored Nodes = 24
Total moves = 9

5.2 Manhattan Distance:

Initial:

1 8 2
0 4 3
7 6 5

Goal:

1 2 3
4 5 6
7 8 0

Heuristic: Manhattan Distance

Leaves:

0 8 2
1 4 3
7 6 5
f(n)=11, g(n)= 1, h(n)= 10

1 8 2
4 0 3
7 6 5
f(n)=9, g(n)= 1, h(n)= 8

1 8 2
7 4 3
0 6 5
f(n)=11, g(n)= 1, h(n)= 10

Leaves:

1 0 2
4 8 3
7 6 5
f(n)=9, g(n)= 2, h(n)= 7

1 8 2
0 4 3
7 6 5
Visited

1 8 2
4 3 0
7 6 5
f(n)=11, g(n)= 2, h(n)= 9

1 8 2
4 6 3
7 0 5
f(n)=9, g(n)= 2, h(n)= 7

Leaves:
0 1 2
4 8 3
7 6 5
f(n)=11, g(n)= 3, h(n)= 8

1 2 0
4 8 3
7 6 5
f(n)=9, g(n)= 3, h(n)= 6

1 8 2
4 0 3
7 6 5
Visited

Leaves:

1 8 2
4 0 3
7 6 5
Visited

1 8 2
4 6 3
0 7 5
f(n)=11, g(n)= 3, h(n)= 8

1 8 2
4 6 3
7 5 0
f(n)=9, g(n)= 3, h(n)= 6

Leaves:

1 0 2
4 8 3
7 6 5
Visited

1 2 3
4 8 0
7 6 5
f(n)=9, g(n)= 4, h(n)= 5

Leaves:

1 8 2
4 6 0
7 5 3
f(n)=11, g(n)= 4, h(n)= 7

1 8 2
4 6 3
7 0 5
Visited

Leaves:

1 2 0
4 8 3
7 6 5
Visited

1 2 3
4 0 8
7 6 5
f(n)=11, g(n)= 5, h(n)= 6

1 2 3
4 8 5
7 6 0
f(n)=9, g(n)= 5, h(n)= 4

Leaves:

1 2 3
4 8 0
7 6 5
Visited

1 2 3
4 8 5
7 0 6
f(n)=9, g(n)= 6, h(n)= 3

Leaves:

1 2 3
4 0 5
7 8 6
f(n)=9, g(n)= 7, h(n)= 2

1 2 3
4 8 5
0 7 6
f(n)=11, g(n)= 7, h(n)= 4

1 2 3
4 8 5
7 6 0
Visited

Leaves:

1 0 3
4 2 5
7 8 6
f(n)=11, g(n)= 8, h(n)= 3

1 2 3
0 4 5
7 8 6
f(n)=11, g(n)= 8, h(n)= 3

1 2 3
4 5 0
7 8 6
f(n)=9, g(n)= 8, h(n)= 1

1 2 3
4 8 5
7 0 6
Visited

Leaves:

1 2 0
4 5 3
7 8 6
f(n)=11, g(n)= 9, h(n)= 2

1 2 3
4 0 5
7 8 6
Visited

1 2 3
4 5 6
7 8 0
f(n)=9, g(n)= 9, h(n)= 0

Path:

1 8 2
0 4 3
7 6 5

1 8 2
4 0 3
7 6 5

1 0 2
4 8 3
7 6 5

1 2 0
4 8 3
7 6 5

1 2 3
4 8 0
7 6 5

1 2 3
4 8 5
7 6 0

1 2 3
4 8 5
7 0 6

1 2 3
4 0 5
7 8 6

1 2 3
4 5 0
7 8 6

1 2 3
4 5 6
7 8 0

Generated Nodes = 23
Explored Nodes = 11
Total moves = 9

Case 6

Initial:

1 2 3
7 4 5
6 8 0

Goal:

1 2 3
8 6 4
7 5 0

6.1 Misplaced tiles:

Heuristic: Misplaced Tiles

Initial:

1 2 3
7 4 5
6 8 0

Goal:

1 2 3
8 6 4
7 5 0

Leaves:

1 2 3
7 4 0
6 8 5
f(n) = 6, g(n) =1, h(n) = 5

1 2 3
7 4 5
6 0 8
f(n) = 6, g(n) = 1, h(n) = 5

Leaves:

1 2 0
7 4 3
6 8 5
f(n)=8, g(n)= 2, h(n)= 6

1 2 3
7 0 4
6 8 5
f(n)=6, g(n)= 2, h(n)= 4

1 2 3
7 4 5
6 8 0
Visited

Leaves:

1 2 3
7 0 5
6 4 8
f(n)=7, g(n)= 2, h(n)= 5

1 2 3
7 4 5
0 6 8
f(n)=7, g(n)= 2, h(n)= 5

1 2 3
7 4 5
6 8 0
Visited

Leaves:

1 0 3
7 2 4
6 8 5
f(n)=8, g(n)= 3, h(n)= 5

1 2 3
0 7 4
6 8 5
f(n)=7, g(n)= 3, h(n)= 4

1 2 3
7 8 4
6 0 5
f(n)=7, g(n)= 3, h(n)= 4

1 2 3
7 4 0
6 8 5
Visited

Leaves:

1 0 3
7 2 5
6 4 8
f(n)=9, g(n)= 3, h(n)= 6

1 2 3
0 7 5
6 4 8
f(n)=8, g(n)= 3, h(n)= 5

1 2 3
7 5 0
6 4 8
f(n)=8, g(n)= 3, h(n)= 5

1 2 3
7 4 5
6 0 8
Visited

Leaves:

1 2 3
0 4 5
7 6 8
f(n)=7, g(n)= 3, h(n)= 4

1 2 3
7 4 5
6 0 8
Visited

Leaves:

0 2 3
1 7 4
6 8 5
f(n)=9, g(n)= 4, h(n)= 5

1 2 3
7 0 4
6 8 5
Visited

1 2 3
6 7 4
0 8 5
f(n)=8, g(n)= 4, h(n)= 4

Leaves:

1 2 3
7 0 4
6 8 5
Visited

1 2 3
7 8 4
0 6 5
f(n)=8, g(n)= 4, h(n)= 4

1 2 3
7 8 4

6 5 0
f(n)=7, g(n)= 4, h(n)= 3

Leaves:

0 2 3
1 4 5
7 6 8
f(n)=9, g(n)= 4, h(n)= 5

1 2 3
4 0 5
7 6 8
f(n)=8, g(n)= 4, h(n)= 4

1 2 3
7 4 5
0 6 8
Visited

Leaves:

1 2 3
7 8 0
6 5 4
f(n)=9, g(n)= 5, h(n)= 4

1 2 3
7 8 4
6 0 5
Visited

Leaves:

1 0 2
7 4 3
6 8 5
f(n)=10, g(n)= 3, h(n)= 7

1 2 3
7 4 0
6 8 5
Visited

Leaves:

0 1 3
7 2 4
6 8 5
f(n)=10, g(n)= 4, h(n)= 6

1 3 0
7 2 4
6 8 5
f(n)=10, g(n)= 4, h(n)= 6

1 2 3
7 0 4
6 8 5
Visited

Leaves:

0 2 3
1 7 5
6 4 8
f(n)=10, g(n)= 4, h(n)= 6

1 2 3
7 0 5
6 4 8
Visited

1 2 3
6 7 5
0 4 8
f(n)=9, g(n)= 4, h(n)= 5

Leaves:

1 2 0
7 5 3
6 4 8
f(n)=10, g(n)= 4, h(n)= 6

1 2 3
7 0 5
6 4 8
Visited

1 2 3
7 5 8
6 4 0
f(n)=9, g(n)= 4, h(n)= 5

Leaves:

1 2 3
0 7 4
6 8 5
Visited

1 2 3
6 7 4
8 0 5
f(n)=9, g(n)= 5, h(n)= 4

Leaves:

1 2 3
0 8 4
7 6 5
f(n)=8, g(n)= 5, h(n)= 3

1 2 3
7 8 4
6 0 5
Visited

Leaves:

1 0 3
4 2 5
7 6 8
f(n)=10, g(n)= 5, h(n)= 5

1 2 3
0 4 5
7 6 8
Visited

1 2 3
4 5 0
7 6 8
f(n)=9, g(n)= 5, h(n)= 4

1 2 3
4 6 5
7 0 8
f(n)=8, g(n)= 5, h(n)= 3

Leaves:

0 2 3
1 8 4
7 6 5
f(n)=10, g(n)= 6, h(n)= 4

1 2 3
8 0 4
7 6 5

f(n)=8, g(n)= 6, h(n)= 2

1 2 3
7 8 4
0 6 5
Visited

Leaves:

1 2 3
4 0 5
7 6 8
Visited

1 2 3
4 6 5
0 7 8
f(n)=10, g(n)= 6, h(n)= 4

1 2 3
4 6 5
7 8 0
f(n)=9, g(n)= 6, h(n)= 3

Leaves:

1 0 3
8 2 4
7 6 5
f(n)=10, g(n)= 7, h(n)= 3

1 2 3
0 8 4
7 6 5
Visited

1 2 3
8 4 0
7 6 5
f(n)=10, g(n)= 7, h(n)= 3

1 2 3
8 6 4
7 0 5
f(n)=8, g(n)= 7, h(n)= 1

Leaves:

1 2 3
8 0 4
7 6 5
Visited

1 2 3
8 6 4
0 7 5
f(n)=10, g(n)= 8, h(n)= 2

1 2 3
8 6 4
7 5 0
f(n)=8, g(n)= 8, h(n)= 0

Final Path:

1 2 3
7 4 5
6 8 0

1 2 3
7 4 0
6 8 5

1 2 3
7 0 4
6 8 5

1 2 3
7 8 4
6 0 5

1 2 3
7 8 4
0 6 5

1 2 3
0 8 4
7 6 5

1 2 3
8 0 4
7 6 5

1 2 3
8 6 4
7 0 5

1 2 3
8 6 4
7 5 0

Generated Nodes = 42
Explored Nodes = 20
Total moves = 8

6.2 Manhattan Distance:

Initial:

1 2 3
7 4 5
6 8 0

Goal:

1 2 3
8 6 4
7 5 0

Leaves:

1 2 3
7 4 0
6 8 5
f(n)=8, g(n)= 1, h(n)= 7

1 2 3
7 4 5
6 0 8
f(n)=10, g(n)= 1, h(n)= 9

Leaves:

1 2 0
7 4 3
6 8 5
f(n)=10, g(n)= 2, h(n)= 8

1 2 3
7 0 4
6 8 5
f(n)=8, g(n)= 2, h(n)= 6

1 2 3
7 4 5
6 8 0
Visited

Leaves:

1 0 3
7 2 4
6 8 5
f(n)=10, g(n)= 3, h(n)= 7

1 2 3
0 7 4
6 8 5
f(n)=10, g(n)= 3, h(n)= 7

1 2 3
7 4 0
6 8 5
Visited

1 2 3
7 8 4
6 0 5
f(n)=8, g(n)= 3, h(n)= 5

Leaves:

1 2 3
7 0 4
6 8 5
Visited

1 2 3
7 8 4
0 6 5
f(n)=8, g(n)= 4, h(n)= 4

1 2 3
7 8 4
6 5 0
f(n)=8, g(n)= 4, h(n)= 4

Leaves:

1 2 3
0 8 4
7 6 5
f(n)=8, g(n)= 5, h(n)= 3

1 2 3
7 8 4
6 0 5
Visited

Leaves:

1 2 3
7 8 0
6 5 4
f(n)=10, g(n)= 5, h(n)= 5

1 2 3
7 8 4
6 0 5
Visited

Leaves:

0 2 3
1 8 4
7 6 5
f(n)=10, g(n)= 6, h(n)= 4

1 2 3
8 0 4
7 6 5
f(n)=8, g(n)= 6, h(n)= 2

1 2 3
7 8 4
0 6 5
Visited

Leaves:

1 0 3
8 2 4
7 6 5
f(n)=10, g(n)= 7, h(n)= 3

1 2 3
0 8 4
7 6 5
Visited

1 2 3
8 4 0
7 6 5
f(n)=10, g(n)= 7, h(n)= 3

1 2 3
8 6 4
7 0 5
f(n)=8, g(n)= 7, h(n)= 1

Leaves:

1 2 3
8 0 4
7 6 5
Visited

1 2 3
8 6 4
0 7 5
f(n)=10, g(n)= 8, h(n)= 2

1 2 3
8 6 4
7 5 0
f(n)=8, g(n)= 8, h(n)= 0

Path:

1 2 3
7 4 5
6 8 0

1 2 3
7 4 0
6 8 5

1 2 3
7 0 4
6 8 5

1 2 3
7 8 4
6 0 5

1 2 3
7 8 4
0 6 5

1 2 3
0 8 4
7 6 5

1 2 3
8 0 4
7 6 5

1 2 3
8 6 4
7 0 5

1 2 3
8 6 4
7 5 0

Generated Nodes = 19
Explored Nodes = 9
Total moves = 8

8 Summary

The below table shows the number of generated and expanded nodes for each case using Misplaced tiles approach and Manhattan distance approach.

Case #	**Initial State**	**Goal State**	**Misplaced Tiles**		**Manhattan Distance**	
			Generated Nodes	**Explored Nodes**	**Generated Nodes**	**Explored Nodes**
Case 1	2 8 1 3 4 6 7 5 0	3 2 1 8 0 4 7 5 6	15	7	13	6
Case 2	1 2 3 5 6 0 7 8 4	1 2 3 5 8 6 0 7 4	9	5	9	5
Case 3	0 1 3 4 2 5 7 8 6	1 2 3 4 5 6 7 8 0	10	5	10	5
Case 4	2 8 3 1 6 4 7 0 5	1 2 3 8 0 4 7 6 5	14	7	12	6
Case 5	1 8 2 0 4 3 7 6 5	1 2 3 4 5 6 7 8 0	54	24	23	11
Case 6	1 2 3 7 4 5 6 8 0	1 2 3 8 6 4 7 5 0	42	20	19	9